Java의 정석

제 14 장

람다와 스트림
(Lambda & Stream)

2016.3.31

남궁성 강의

castello@naver.com

1.1 람다식(Lambda Expression)이란?

▶ 함수(메서드)를 간단한 '식(Expression)'으로 표현하는 방법

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

→

```
(a, b) -> a > b ? a : b
```

▶ 익명 함수(이름이 없는 함수, anonymous function)

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

→

```
int max(int a, int b) -> {
    return a > b ? a : b;
}
```

▶ 함수와 메서드의 차이

- 근본적으로 동일. 함수는 일반적 용어, 메서드는 객체지향개념 용어
- 함수는 클래스에 독립적, 메서드는 클래스에 종속적

1.2 람다식 작성하기

1. 메서드의 이름과 반환타입을 제거하고 '->'를 블록{} 앞에 추가한다.

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

```
int max(int a, int b) -> {
    return a > b ? a : b;
}
```

2. 반환값이 있는 경우, 식이나 값만 적고 return문 생략 가능(끝에';'안 붙임)

```
(int a, int b) -> {
    return a > b ? a : b;
}
```

```
(int a, int b)-> a > b ? a : b
```

3. 매개변수의 타입이 추론 가능하면 생략가능(대부분의 경우 생략가능)

```
(int a, int b)-> a > b ? a : b
```

```
(a, b)-> a > b ? a : b
```

1.2 람다식 작성하기 - 주의사항

1. 매개변수가 하나인 경우, 괄호() 생략가능(타입이 없을 때만)

```
(a) -> a * a
(int a)-> a * a
```

```
a -> a * a  // OK
int a -> a * a  // 에러
```

2. 블록 안의 문장이 하나뿐 일 때, 괄호{}생략가능(끝에 ';' 안 붙임)

```
(int i) -> {
  System.out.println(i);
}
```

```
(int i)-> System.out.println(i)
```

단, 하나뿐인 문장이 return문이면 괄호{} 생략불가

```
(int a, int b) -> { return a > b ? a : b; }   // OK
(int a, int b) ->   return a > b ? a : b      // 에러
```

1.2 람다식 작성하기 - 실습

메서드	람다식

`int max(int a, int b) {`
`  return a > b ? a : b;`
`}`
	①

`int printVar(String name, int i) {`
`  System.out.println(name+"="+i);`
`}`
	②

`int square(int x) {`
`  return x * x;`
`}`
	③

`int roll() {`
`  return (int)(Math.random()*6);`
`}`
	④

1.3 함수형 인터페이스(1/3)

▶ 람다식은 익명 함수? 사실은 익명 객체!!!

```
(a, b)-> a > b ? a : b
```

```
new Object() {
    int max(int a, int b) {
        return a > b ? a : b;
    }
}
```

▶ 람다식(익명 객체)을 다루기 위한 참조변수가 필요. 참조변수의 타입은?

```
Object obj = new Object() {
    int max(int a, int b) {
        return a > b ? a : b;
    }
};
```

```
타입 obj = (a, b)-> a > b ? a : b ; // 어떤 타입?
```

```
int value = obj.max(3,5); // 에러. Object클래스에 max()가 없음
```

1.3 함수형 인터페이스(2/3)

▶ 함수형 인터페이스 – 단 하나의 추상 메서드만 선언된 인터페이스

```
interface MyFunction {
    public abstract int max(int a, int b);
}
```

```
MyFunction f = new MyFunction() {
                    public int max(int a, int b) {
                        return a > b ? a : b;
                    }
                };
```

```
int value = f.max(3,5); // OK. MyFunction에 max()가 있음
```

▶ 함수형 인터페이스 타입의 참조변수로 람다식을 참조할 수 있음.
(단, 함수형 인터페이스의 메서드와 람다식의 **매개변수 개수와 반환타입**이 일치해야 함.)

```
MyFunction f = (a, b) -> a > b ? a : b;

int value = f.max(3,5); // 실제로는 람다식(익명 함수)이 호출됨
```

1.3 함수형 인터페이스 - example

▶ 익명 객체를 람다식으로 대체

```
List<String> list = Arrays.asList("abc", "aaa", "bbb", "ddd", "aaa");

Collections.sort(list, new Comparator<String>() {
                           public int compare(String s1, String s2) {
                               return s2.compareTo(s1);
                           }
                       });
```

```
interface Comparator<T> {
   int compare(T o1, T o2);
}
```

```
List<String> list = Arrays.asList("abc", "aaa", "bbb", "ddd", "aaa");
Collections.sort(list,(s1,s2)-> s2.compareTo(s1));
```

1.3 함수형 인터페이스 (3/3)– 매개변수와 반환타입

```
@FunctionalInterface
interface MyFunction {
    void myMethod();
}
```

▶ 함수형 인터페이스 타입의 매개변수

```
void aMethod(MyFunction f) {
    f.myMethod(); // MyFunction에 정의된 메서드 호출
}
```

```
MyFunction f = ()-> System.out.println("myMethod()");
aMethod(f);
```

```
aMethod(()-> System.out.println("myMethod()"));
```

▶ 함수형 인터페이스 타입의 반환타입

```
MyFunction myMethod() {
    MyFunction f = ()->{};
    return f;
}
```

```
MyFunction myMethod() {
    return ()->{};
}
```

1.4 java.util.function패키지(1/5)

▶ 자주 사용되는 다양한 함수형 인터페이스를 제공.

함수형 인터페이스	메서드	설 명
java.lang. Runnable	void run()	매개변수도 없고, 반환값도 없음.
Supplier<T>	T get() → T	매개변수는 없고, 반환값만 있음.
Consumer<T>	T → void accept(T t)	Supplier와 반대로 매개변수만 있고, 반환값이 없음
Function<T,R>	T → R apply(T t) → R	일반적인 함수. 하나의 매개변수를 받아서 결과를 반환
Predicate<T>	T → boolean test(T t) → boolean	조건식을 표현하는데 사용됨. 매개변수는 하나, 반환 타입은 boolean

```
Predicate<String> isEmptyStr = s -> s.length()==0;
String s = "";

if(isEmptyStr.test(s)) // if(s.length()==0)
    System.out.println("This is an empty String.");
```

1.4 java.util.function패키지 - Quiz

Q. 아래의 빈 칸에 알맞은 함수형 인터페이스(java.util.function패키지)를 적으시오.

```
[      ①    ]  f = ()-> (int)(Math.random()*100)+1;
[      ②    ]  f = i -> System.out.print(i+", ");
[      ③    ]  f = i -> i%2==0;
[      ④    ]  f = i -> i/10*10;
```

1.4 java.util.function패키지(2/5)

▶ 매개변수가 2개인 함수형 인터페이스

함수형 인터페이스	메서드	설 명
BiConsumer<T,U>	T, U → void accept(T t, U u)	두개의 매개변수만 있고, 반환값이 없음
BiPredicate<T,U>	T, U → boolean test(T t, U u) → boolean	조건식을 표현하는데 사용됨. 매개변수는 둘, 반환값은 boolean
BiFunction<T,U,R>	T, U → R apply(T t, U u) → R	두 개의 매개변수를 받아서 하나의 결과를 반환

```
@FunctionalInterface
interface TriFunction<T,U,V,R> {
   R apply(T t, U u, V v);
}
```

1.4 java.util.function패키지(3/5)

▶ 매개변수의 타입과 반환타입이 일치하는 함수형 인터페이스

함수형 인터페이스	메서드	설 명
UnaryOperator<T>	T → T apply(T t) → T	Function의 자손, Function과 달리 매개변수와 결과의 타입이 같다.
BinaryOperator<T>	T, T → T apply(T t, T t) → T	BiFunction의 자손, BiFunction과 달리 매개변수와 결과의 타입이 같다.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,T> {
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

```
@FunctionalInterface
public interface Function<T,R> {
    R apply(T t);
    ...
}
```

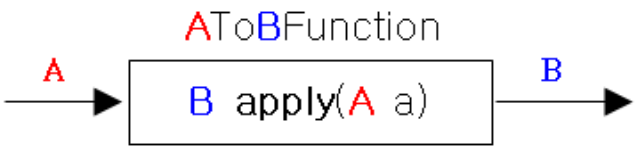
1.4 java.util.function패키지(4/5)

▶ 함수형 인터페이스를 사용하는 컬렉션 프레임웍의 메서드

인터페이스	메서드	설명
Collection	boolean removeIf(Predicate<E> filter)	조건에 맞는 요소를 삭제
List	void replaceAll(UnaryOperator<E> operator)	모든 요소를 변환하여 대체
Iterable	void forEach(Consumer<T> action)	모든 요소에 작업 action을 수행
Map	V compute(K key, BiFunction<K,V,V> f)	지정된 키의 값에 작업 f를 수행
	V computeIfAbsent(K key, Function<K,V> f)	키가 없으면, 작업 f 수행 후 추가
	V computeIfPresent(K key, BiFunction<K,V,V> f)	지정된 키가 있을 때,작업 f 수행
	V merge(K key, V value, BiFunction<V,V,V> f)	모든 요소에 병합작업 f를 수행
	void forEach(BiConsumer<K,V> action)	모든 요소에 작업 action을 수행
	void replaceAll(BiFunction<K,V,V> f)	모든 요소에 치환작업 f를 수행

```
list.forEach(i->System.out.print(i+","));  // list의 모든 요소를 출력
list.removeIf(x-> x%2==0 || x%3==0);       // 2 또는 3의 배수를 제거
list.replaceAll(i->i*10);                  // 모든 요소에 10을 곱한다.
// map의 모든 요소를 {k,v}의 형식으로 출력
map.forEach((k,v)-> System.out.print("{"+k+","+v+"},"));
```

1.4 java.util.function패키지(5/5)



▶ 기본형을 사용하는 함수형 인터페이스

함수형 인터페이스	메서드	설 명
DoubleToIntFunction	double → int applyAsInt(double d) → int	AToBFunction은 입력이 A타입 출력이 B타입
ToIntFunction<T>	T → int applyAsInt(T value) → int	ToBFunction은 출력이 B타입이다. 입력은 지네릭 타입
IntFunction<R>	int → R apply(int value) → R	AFunction은 입력이 A타입이고 출력은 지네릭 타입
ObjIntCunsumer<T>	T, int → void accept(T t, int i)	ObjAFunction은 입력이 T, int 타입이고 출력은 없다.

```
Supplier<Integer> s = ()->(int)(Math.random()*100)+1;

static <T> void makeRandomList(Supplier<T> s, List<T> list) {
    for(int i=0;i<10;i++)
        list.add(s.get()); // List<Integer> list = new ArrayList<>();
```

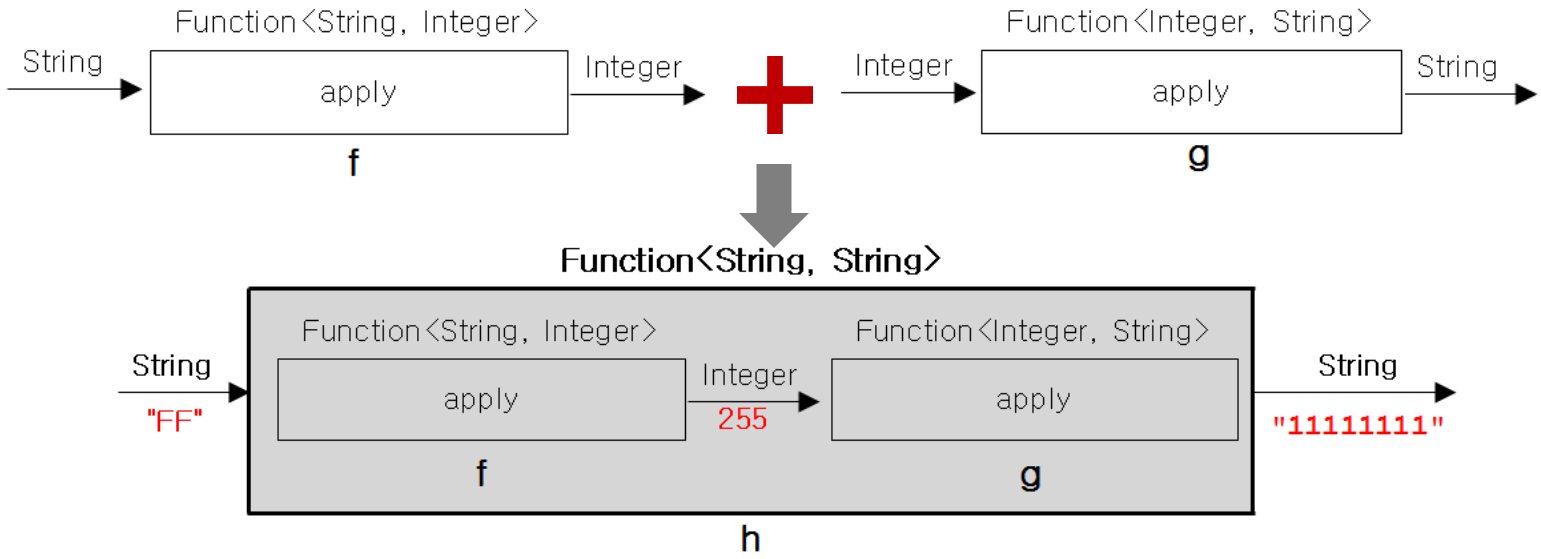
```
IntSupplier s = ()->(int)(Math.random()*100)+1;

static void makeRandomList(IntSupplier s, int[] arr) {
    for(int i=0;i<arr.length;i++)
        arr[i] = s.getAsInt();   // get()이 아니라 getAsInt()임에 주의
```

1.5 Function의 합성(1/2)

▶ Function타입의 두 람다식을 하나로 합성 – andThen()

```
Function<String, Integer> f = (s) -> Integer.parseInt(s, 16); // s를 16진 정수로 변환
Function<Integer, String> g = (i) -> Integer.toBinaryString(i); // 2진 문자열로 변환
Function<String, String>  h =  f.andThen(g);   // f + g → h
```

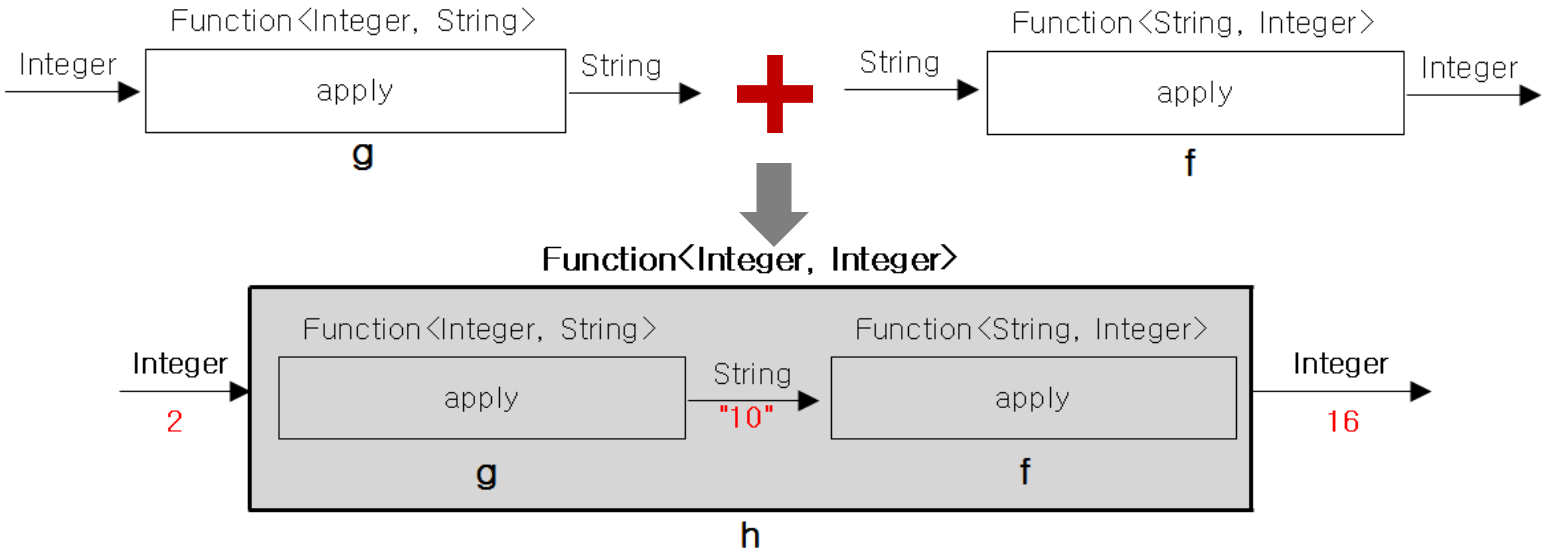



```
System.out.println(h.apply("FF")); // "FF" → 255 → "11111111"
```

1.5 Function의 합성(2/2)

▶ Function타입의 두 람다식을 하나로 합성 – compose()

```
Function<Integer, String> g = (i) -> Integer.toBinaryString(i); // 2진 문자열로 변환
Function<String, Integer> f = (s) -> Integer.parseInt(s, 16);// s를 16진 정수로 변환
Function< Integer, Integer >  h =  f.compose(g);   // g + f → h
```




```
System.out.println(h.apply(2)); // 2 → "10" → 16
```

1.6 Predicate의 결합

▶ and(), or(), negate()로 두 Predicate를 하나로 결합(default메서드)

```
Predicate<Integer> p = i -> i < 100;
Predicate<Integer> q = i -> i < 200;
Predicate<Integer> r = i -> i%2 == 0;
```

```
Predicate<Integer> notP = p.negate();        // i >= 100
Predicate<Integer> all = notP.and(q).or(r);  // 100 <= i && i < 200 || i%2==0
Predicate<Integer> all2 = notP.and(q.or(r)); // 100 <= i && (i < 200 || i%2==0)
```

```
System.out.println(all.test(2));   // true
System.out.println(all2.test(2));  // false
```

▶ 등가비교를 위한 Predicate의 작성에는 isEqual()를 사용(static메서드)

```
Predicate<String> p = Predicate.isEqual(str1);  // isEquals()은 static메서드
Boolean result = p.test(str2);  // str1과 str2가 같은지 비교한 결과를 반환
```

```
boolean result = Predicate.isEqual(str1).test(str2);
```

1.7 메서드 참조(method reference)(1/2)

▶ 하나의 메서드만 호출하는 람다식은 '메서드 참조'로 간단히 할 수 있다.

종류	람다식	메서드 참조
static메서드 참조	(x) -> ClassName.method(x)	ClassName::method
인스턴스메서드 참조	(obj, x) -> obj.method(x)	ClassName::method
특정 객체 인스턴스메서드 참조	(x) -> obj.method(x)	obj::method

▶ static메서드 참조

```
Integer method(String s) {  // 그저 Integer.parseInt(String s)만 호출
    return Integer.parseInt(s);
}
```

```
int result = obj.method("123");
int result = Integer.parseInt("123");
```

```
Function<String, Integer> f = (String s) -> Integer.parseInt(s);
```

```
Function<String, Integer> f = Integer::parseInt;  // 메서드 참조
```

1.7 메서드 참조(method reference)(2/2)

▶ 인스턴스 메서드 참조

```
BiFunction<String,String,Boolean> f = (s1, s2) -> s1.equals(s2);
```

```
BiFunction<String,String,Boolean> f = String::equals;
```

▶ 특정 객체의 인스턴스 메서드 참조

```
MyClass obj = new MyClass();
Function<String, Boolean> f  = (x) -> obj.equals(x); // 람다식
Function<String, Boolean> f2 = obj::equals;          // 메서드 참조
```

▶ new연산자(생성자, 배열)와 메서드 참조

```
Supplier<MyClass> s = MyClass::new;           // ()  -> new MyClass()
Function<Integer, MyClass> f2 = MyClass::new; // (i) -> new MyClass(i)

Function<Integer, int[]> f2 = int[]::new;     // x -> new int[x];
```

2.1 스트림(Stream)이란?

▶ 다양한 데이터 소스를 표준화된 방법으로 다루기 위한 것

Stream<T> Collection.stream()

```
List<Integer> list = Arrays.asList(1,2,3,4,5);
Stream<Integer> intStream  = list.stream(); // 컬렉션.
Stream<String>  strStream  = Stream.of(new String[]{"a","b","c"}); // 배열
Stream<Integer> evenStream   = Stream.iterate(0, n->n+2);  // 0,2,4,6, ...
Stream<Double>  randomStream = Stream.generate(Math::random);     // 람다식
IntStream       intStream    = new Random().ints(5); // 난수 스트림(크기가 5)
```

▶ 스트림이 제공하는 기능 – 중간 연산과 최종 연산

- 중간 연산 - 연산결과가 스트림인 연산. 반복적으로 적용가능
- 최종 연산 - 연산결과가 스트림이 아닌 연산. 스트림의 요소를 소모하므로 한번만 적용가능

```
stream.distinct().limit(5).sorted().forEach(System.out::println)
       중간 연산    중간 연산   중간 연산          최종 연산
```

```
String[] strArr = { "dd","aaa","CC","cc","b" };
Stream<String> stream = Stream.of(strArr); // 문자열 배열이 소스인 스트림
Stream<String> filteredStream   = stream.filter();   // 걸러내기(중간 연산)
Stream<String> distinctedStream = stream.distinct(); // 중복제거(중간 연산)
Stream<String> sortedStream     = stream.sort();     // 정렬(중간 연산)
Stream<String> limitedStream    = stream.limit(5); // 스트림 자르기(중간 연산)
int total = stream.count(); // 요소 개수 세기(최종연산)
```

2.2 스트림(Stream)의 특징(1/2)

▶ 스트림은 데이터 소스로부터 데이터를 읽기만할 뿐 변경하지 않는다.

```
List<Integer> list = Arrays.asList(3,1,5,4,2);
List<Integer> sortedList = list.stream().sorted()    // list를 정렬해서
                        .collect(Collectors.toList()); // 새로운 List에 저장
System.out.println(list);         // [3, 1, 5, 4, 2]
System.out.println(sortedList);   // [1, 2, 3, 4, 5]
```

▶ 스트림은 Iterator처럼 일회용이다.(필요하면 다시 스트림을 생성해야 함)

```
strStream.forEach(System.out::println); // 모든 요소를 화면에 출력(최종연산)
int numOfStr = strStream.count();       // 에러. 스트림이 이미 닫혔음.
```

▶ 최종 연산 전까지 중간연산이 수행되지 않는다. – 지연된 연산

```
IntStream intStream = new Random().ints(1,46);   // 1~45범위의 무한 스트림
intStream.distinct().limit(6).sorted()           // 중간 연산
         .forEach(i->System.out.print(i+","));   // 최종 연산
```

2.2 스트림(Stream)의 특징(2/2)

▶ 스트림은 작업을 내부 반복으로 처리한다.

```
for(String str : strList)
    System.out.println(str);
```

```
stream.forEach(System.out::println);
```

```
void forEach(Consumer<? super T> action)  {
        Objects.requireNonNull(action);   // 매개변수의 널 체크

        for(T t : src)  // 내부 반복(for문을 메서드 안으로 넣음)
            action.accept(T);
}
```

▶ 스트림의 작업을 병렬로 처리 – 병렬스트림

```
Stream<String> strStream = Stream.of("dd","aaa","CC","cc","b");
int sum = strStream.parallelStream() // 병렬 스트림으로 전환(속성만 변경)
                   .mapToInt(s -> s.length()).sum(); // 모든 문자열의 길이의 합
```

▶ 기본형 스트림 – IntStream, LongStream, DoubleStream

- 오토박싱&언박싱의 비효율이 제거됨(Stream<Integer>대신 IntStream사용)
- 숫자와 관련된 유용한 메서드를 Stream<T>보다 더 많이 제공

2.3 스트림의 생성(1/3)

▶ 컬렉션으로부터 스트림 생성하기

```
List<Integer> list = Arrays.asList(1,2,3,4,5);
Stream<Integer> intStream = list.stream();// Stream<T> Collection.stream()
```

▶ 배열로부터 스트림 생성하기

```
Stream<String> strStream = Stream.of("a","b","c"); // 가변 인자
Stream<String> strStream = Stream.of(new String[]{"a","b","c"});
Stream<String> strStream = Arrays.stream(new String[]{"a","b","c"});
Stream<String> strStream = Arrays.stream(new String[]{"a","b","c"}, 0, 3);
```

▶ 특정 범위의 정수를 요소로 갖는 스트림 생성하기

```
IntStream intStream = IntStream.range(1, 5);       // 1,2,3,4
IntStream intStream = IntStream.rangeClosed(1, 5); // 1,2,3,4,5
```

2.3 스트림의 생성(2/3)

▶ 난수를 요소로 갖는 스트림 생성하기

```
IntStreamintStream = new Random().ints();         // 무한 스트림
intStream.limit(5).forEach(System.out::println); // 5개의 요소만 출력한다.

IntStream intStream = new Random().ints(5);  // 크기가 5인 난수 스트림을 반환
```

```
Integer.MIN_VALUE <=  ints()  <= Integer.MAX_VALUE
   Long.MIN_VALUE <=  longs() <= Long.MAX_VALUE
              0.0 <= doubles()< 1.0
```

* 지정된 범위의 난수를 요소로 갖는 스트림을 생성하는 메서드

```
IntStream    ints(int begin, int end)                    // 무한 스트림
LongStream   longs(long begin, long end)
DoubleStream doubles(double begin, double end)

IntStream    ints(long streamSize, int begin, int end)   // 유한 스트림
LongStream   longs(long streamSize, long begin, long end)
DoubleStream doubles(long streamSize, double begin, double end)
```

2.3 스트림의 생성(3/3)

▶ 람다식을 소스로 하는 스트림 생성하기

```
static <T> Stream<T> iterate(T seed,UnaryOperator<T> f) // 이전 요소에 종속적
static <T> Stream<T> generate(Supplier<T> s)            // 이전 요소에 독립적
```

```
Stream<Integer> evenStream   = Stream.iterate(0, n->n+2);  // 0,2,4,6, ...

Stream<Double>  randomStream = Stream.generate(Math::random);
Stream<Integer> oneStream    = Stream.generate(()->1);
```

▶ 파일을 소스로 하는 스트림 생성하기

```
Stream<Path>   Files.list(Path dir)   // Path는 파일 또는 디렉토리
```

```
Stream<String> Files.lines(Path path)
Stream<String> Files.lines(Path path, Charset cs)
Stream<String> lines() // BufferedReader클래스의 메서드
```

2.4 스트림의 중간연산(1/6)

```
IntStream skip(long n)
IntStream limit(long maxSize)
```

▶ 스트림 자르기 – skip(), limit()

```
Stream<T> skip(long n)         // 앞에서부터 n개 건너뛰기
Stream<T> limit(long maxSize) // maxSize 이후의 요소는 잘라냄
```

```
IntStream intStream = IntStream.rangeClosed(1, 10);    // 12345678910
intStream.skip(3).limit(5).forEach(System.out::print); // 45678
```

▶ 스트림의 요소 걸러내기 – filter(), distinct()

```
Stream<T> filter(Predicate<? super T> predicate) // 조건에 맞지 않는 요소 제거
Stream<T> distinct()                             // 중복제거
```

```
IntStream intStream = IntStream.of(1,2,2,3,3,3,4,5,5,6);
intStream.distinct().forEach(System.out::print);          // 123456
```

```
IntStream intStream = IntStream.rangeClosed(1, 10);       // 12345678910
intStream.filter(i->i%2==0).forEach(System.out::print);   // 246810
```

```
intStream.filter(i->i%2!=0 && i%3!=0).forEach(System.out::print);
intStream.filter(i->i%2!=0).filter(i->i%3!=0).forEach(System.out::print);
```

2.4 스트림의 중간연산(2/6)

▶ 스트림 정렬하기 - sorted()

```
Comparator<String> CASE_INSENSITIVE_ORDER
                   = new CaseInsensitiveComparator();
```

```
Stream<T> sorted()                          // 스트림 요소의 기본 정렬(Comparable)로 정렬
Stream<T> sorted(Comparator<? super T> comparator) //지정된 Comparator로 정렬
```

문자열 스트림 정렬 방법 Stream<String> strStream = Stream.of("dd","aaa","CC","cc","b");	출력결과

strStream.sorted() // 기본 정렬
strStream.sorted(Comparator.naturalOrder()) // 기본 정렬
strStream.sorted((s1, s2) -> s1.compareTo(s2)); // 람다식도 가능
strStream.sorted(String::compareTo); // 위의 문장과 동일
	CCaaabccdd

strStream.sorted(Comparator.reverseOrder()) // 기본 정렬의 역순
strStream.sorted(Comparator.<String>naturalOrder().reversed())
	ddccbaaaCC

strStream.sorted(String.CASE_INSENSITIVE_ORDER) // 대소문자 구분안함	aaab**CCcc**dd
strStream.sorted(String.CASE_INSENSITIVE_ORDER.reversed()) // 오타 아님→	dd**CCcc**baaa

strStream.sorted(Comparator.comparing(String::length)) // 길이 순 정렬
strStream.sorted(Comparator.comparingInt(String::length)) // no오토박싱
	bddCCccaaa

strStream.sorted(Comparator.comparing(String::length).reversed())	aaaddCCccb

```
studentStream.sorted(Comparator.comparing(Student::getBan)   // 반별로 정렬
                    .thenComparing(Student::getTotalScore)   // 총점별로 정렬
                    .forEach(Sysetm.out::println);
```

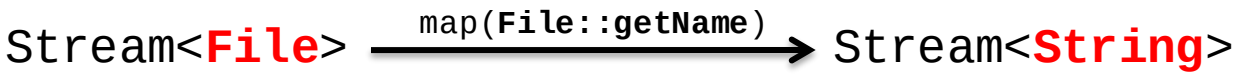
2.4 스트림의 중간연산(3/6)

▶ 스트림의 요소 변환하기 - map()

```
Stream<R> map(Function<? super T,? extends R> mapper) //Stream<T>→Stream<R>
```

```
Stream<File> fileStream = Stream.of(new File("Ex1.java"), new File("Ex1")
          new File("Ex1.bak"), new File("Ex2.java"), new File("Ex1.txt"));

Stream<String> filenameStream = fileStream.map(File::getName);
filenameStream.forEach(System.out::println); // 스트림의 모든 파일의 이름을 출력
```



ex) 파일 스트림(Stream<File>)에서 파일 확장자(대문자)를 중복없이 뽑아내기

```
fileStream.map(File::getName)              // Stream<File> → Stream<String>
   .filter(s->s.indexOf('.')!=-1)          // 확장자가 없는 것은 제외
   .map(s->s.substring(s.indexOf('.')+1)) // Stream<String>→Stream<String>
   .map(String::toUpperCase)               // Stream<String>→Stream<String>
   .distinct() // 중복 제거
   .forEach(System.out::print); // JAVABAKTXT
```

2.4 스트림의 중간연산(4/6)

▶ 스트림을 기본 스트림으로 변환 - mapToInt(), mapToLong(), mapToDouble()

```
IntStream    mapToInt(ToIntFunction<? super T> mapper)        // Stream<T>→IntStream
LongStream   mapToLong(ToLongFunction<? super T> mapper)      // Stream<T>→LongStream
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) // Stream<T>→DoubleStream
```

```
Stream<Integer> studentScoreStream = stuStream.map(Student::getTotalScore);
int sum = studentScoreStream.reduce(0, (a,b)-> a+b);
```

```
IntStream studentScoreStream = studentStream.mapToInt(Student::getTotalScore);
int allTotalScore = studentScoreStream.sum(); // IntStream의 sum()
```

int	sum()
OptionalInt	max()
OptionalInt	min()
OptionalDouble	average()

▶ 기본 스트림을 스트림으로 변환 - mapToObj(), boxed()

```
Stream<T>  mapToObj(IntFunction<? extends T> mapper) // IntStream → Stream<T>
Stream<Integer>  boxed()                             // IntStream → Stream<Integer>
```

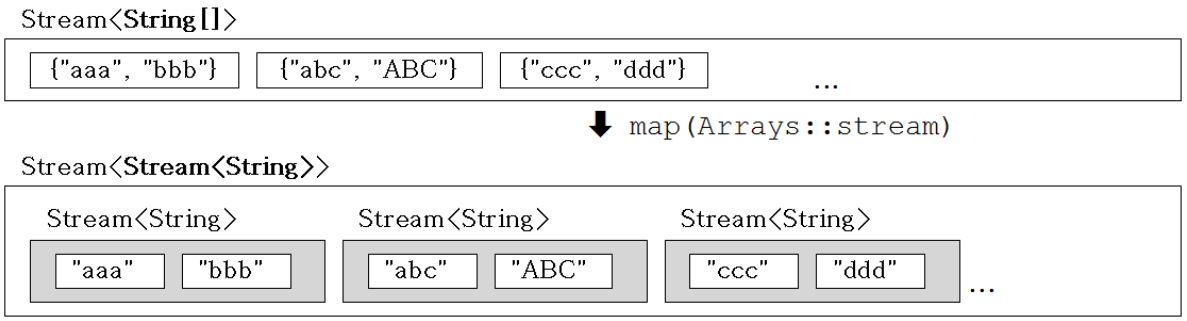
```
IntStream intStream = new Random().ints(1,46); // 1~45사이의 정수(46은 포함안됨)
Stream<Integer> integerStream = intStream.boxed();     // IntStream → Stream<Integer>
Stream<String> lottoStream = intStream.distinct().limit(6).sorted()
                                .mapToObj(i -> i+","); // IntStream → Stream<String>
lottoStream.forEach(System.out::print); // 12,14,20,23,26,29,
```

2.4 스트림의 중간연산(5/6)

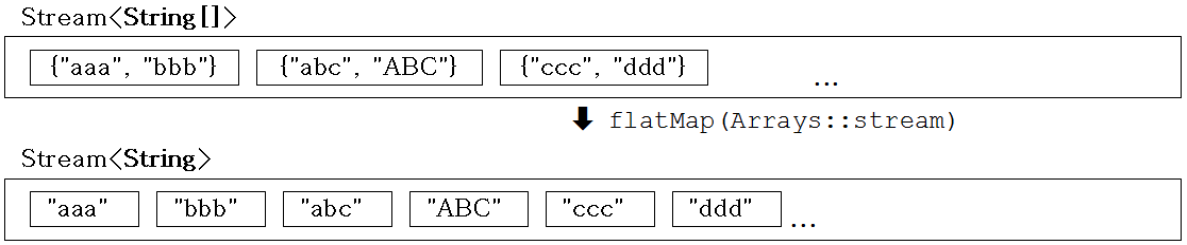
▶ 스트림의 스트림을 스트림으로 변환 – flatMap()

```
Stream<String[]> strArrStrm = Stream.of(new String[]{"abc", "def", "ghi"  },
                                        new String[]{"ABC", "GHI", "JKLMN"});
```

```
Stream<Stream<String>> strStrStrm = strArrStrm.map(Arrays::stream);
```




```
Stream<String> strStrStrm = strArrStrm.flatMap(Arrays::stream); // Arrays.stream(T[])
```



2.4 스트림의 중간연산(6/6)

▶ 스트림의 요소를 소비하지 않고 엿보기 - peek()

```
Stream<T>  peek(Consumer<? super T> action)    // 중간 연산(스트림을 소비X)
void       forEach(Consumer<? super T> action) // 최종 연산(스트림을 소비O)
```

```
fileStream.map(File::getName) // Stream<File> → Stream<String>
    .filter(s -> s.indexOf('.')!=-1) // 확장자가 없는 것은 제외
    .peek(s->System.out.printf("filename=%s%n", s)) // 파일명을 출력한다.
    .map(s -> s.substring(s.indexOf('.')+1)) // 확장자만 추출
    .peek(s->System.out.printf("extension=%s%n", s)) // 확장자를 출력한다.
    .forEach(System.out::println); // 최종연산 스트림을 소비.
```

2.5 Optional<T>과 OptionalInt(1/2)

▶ 'T'타입 객체의 래퍼클래스 - Optional<T>

```
public final class Optional<T> {
    private final T value
    ...
}
```

```
String str = "abc";
Optional<String> optVal = Optional.of(str);
Optional<String> optVal = Optional.of("abc");
Optional<String> optVal = Optional.of(null);         // NullPointerException발생
Optional<String> optVal = Optional.ofNullable(null); // OK
```

▶ Optional객체의 값 가져오기 - get(), orElse(), orElseGet(), orElseThrow()

```
Optional<String> optVal = Optional.of("abc");
String str1 = optVal.get();                    // optVal에 저장된 값을 반환. null이면 예외발생
String str2 = optVal.orElse("");               // optVal에 저장된 값이 null일 때는, ""를 반환
String str3 = optVal.orElseGet(String::new); // 람다식 사용가능 () -> new String()
String str4 = optVal.orElseThrow(NullPointerException::new); // 널이면 예외발생
```

```
T orElseGet(Supplier<? extends T> other)
T orElseThrow(Supplier<? extends X> exceptionSupplier)
```

▶ isPresent() - Optional객체의 값이 null이면 false, 아니면 true를 반환

```
if(Optional.ofNullable(str).isPresent()) {  // if(str!=null) {
    System.out.println(str);
}
```

```
// ifPresnt(Consumer) - 널이 아닐때만 작업 수행, 널이면 아무 일도 안 함
Optional.ofNullable(str).ifPresent(System.out::println);
```

2.5 Optional<T>과 OptionalInt(2/2)

▶ 기본형 값을 감싸는 래퍼클래스 - OptionalInt, OptionalLong, OptionalDouble

```
public final class OptionalInt {
    ...
    private final boolean isPresent; // 값이 저장되어 있으면 true
    private final int value; // int타입의 변수
```

▶ OptionalInt의 값 가져오기 - int getAsInt()

Optional클래스	값을 반환하는 메서드
Optional<T>	T get()
OptionalInt	int getAsInt()
OptionalLong	long getAsLong()
OptionalDouble	double getAsDouble()

▶ 빈 Optional객체의 비교

```
OptionalInt opt1 = OptionalInt.of(0);   // OptionalInt에 0을 저장
OptionalInt opt2 = OptionalInt.empty(); // 빈 OptionalInt객체. OptionalInt에 0이 저장됨
Optional<String> opt3 = Optional.ofNullable(null);  // null이 저장된 Optional객체
Optional<String> opt4 = Optional.empty();           // 빈 Optional객체. null이 저장됨
System.out.println(opt1.equals(opt2)); // false
System.out.println(opt3.equals(opt4)); // true
```

2.6 스트림의 최종연산(1/4)

▶ 스트림의 모든 요소에 지정된 작업을 수행 - forEach(), forEachOrdered()

```
void  forEach(Consumer<? super T> action)        // 병렬스트림인 경우 순서가 보장되지 않음
void  forEachOrdered(Consumer<? super T> action) // 병렬스트림인 경우에도 순서가 보장됨
```

```
IntStream.range(1, 10).sequential().forEach(System.out::print);        // 123456789
IntStream.range(1, 10).sequential().forEachOrdered(System.out::print); // 123456789
```

```
IntStream.range(1, 10).parallel().forEach(System.out::print);          // 683295714
IntStream.range(1, 10).parallel().forEachOrdered(System.out::print);   // 123456789
```

▶ 스트림을 배열로 변환 - toArray()

```
Object[] toArray()                              // 스트림의 모든 요소를 Object배열에 담아 반환
A[]      toArray(IntFunction<A[]> generator) // 스트림의 모든 요소를 A타입의 배열에 담아 반환
```

```
Student[] stuNames = studentStream.toArray(Student[]::new); // OK. x-> new Student[x]
Student[] stuNames = studentStream.toArray(); // 에러.
Object[]  stuNames = studentStream.toArray(); // OK.
```

2.6 스트림의 최종연산(2/4)

▶ 조건 검사 – allMatch(), anyMatch(), noneMatch()

```
boolean allMatch (Predicate<? super T> predicate) // 모든 요소가 조건을 만족시키면 true
boolean anyMatch (Predicate<? super T> predicate) // 한 요소라도 조건을 만족시키면 true
boolean noneMatch(Predicate<? super T> predicate) // 모든 요소가 조건을 만족시키지 않으면 true
```

```
boolean hasFailedStu = stuStream.anyMatch(s-> s.getTotalScore()<=100); // 낙제자가 있는지?
```

▶ 조건에 일치하는 요소 찾기 – findFirst() , findAny()

```
Optional<T> findFirst()     // 첫 번째 요소를 반환. 순차 스트림에 사용
Optional<T> findAny()       // 아무거나 하나를 반환. 병렬 스트림에 사용
```

```
Optional<Student> result = stuStream.filter(s-> s.getTotalScore() <= 100).findFirst();
Optional<Student> result = parallelStream.filter(s-> s.getTotalScore() <= 100).findAny();
```

2.6 스트림의 최종연산(3/4)

▶ 스트림에 대한 통계정보 제공 – count(), sum(), average(), max(), min()

Stream<T>

```
long          count()
Optional<T>   max(Comparator<? super T> comparator)
Optional<T>   min(Comparator<? super T> comparator)
```

IntStream

```
long                  count()
Int                   sum()
OptionalDouble        average()
OptionalInt           max()
OptionalInt           min()
IntSummaryStatistics  summaryStatistics()
```

```
double getAverage()
long   getCount()
int    getMax()
int    getMin()
long   getSum()
```

IntSummaryStatistics

2.6 스트림의 최종연산(4/4)

▶ 스트림의 요소를 하나씩 줄여가며 누적연산 수행 – reduce()

```
Optional<T> reduce(BinaryOperator<T> accumulator)
T           reduce(T identity, BinaryOperator<T> accumulator)
U           reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)
```

- identity - 초기값
- accumulator - 이전 연산결과와 스트림의 요소에 수행할 연산
- combiner - 병렬처리된 결과를 합치는데 사용할 연산(병렬 스트림)

```
int a = identity;

for(int b : stream)
    a = a + b;  // sum()
```

```
// int reduce(int identity, IntBinaryOperator op)
int count = intStream.reduce(0, (a,b) -> a + 1);                        // count()
int sum   = intStream.reduce(0, (a,b) -> a + b);                        // sum()
int max   = intStream.reduce(Integer.MIN_VALUE,(a,b)-> a > b ? a : b); // max()
int min   = intStream.reduce(Integer.MAX_VALUE,(a,b)-> a < b ? a : b); // min()
```

```
// OptionalInt reduce(IntBinaryOperator accumulator)
OptionalInt max = intStream.reduce((a,b) -> a > b ? a : b); // max()
OptionalInt min = intStream.reduce((a,b) -> a < b ? a : b); // min()
```

```
OptionalInt max = intStream.reduce(Integer::max); // static int max(int a, int b)
OptionalInt min = intStream.reduce(Integer::min); // static int min(int a, int b)
```

2.7 collect(), Collector, Collectors

▶ collect()는 Collector를 매개변수로 하는 스트림의 최종연산

```
Object collect(Collector collector) // Collector를 구현한 클래스의 객체를 매개변수로
Object collect(Supplier supplier, BiConsumer accumulator, BiConsumer combiner) // 잘 안쓰임
```

▶ Collector는 수집(collect)에 필요한 메서드를 정의해 놓은 인터페이스

```
public interface Collector<T, A, R> {  // T(요소)를 A에 누적한 다음, 결과를 R로 변환해서 반환
   Supplier<A>           supplier();         // StringBuilder::new            누적할 곳
   BiConsumer<A, T>      accumulator();      // (sb, s) -> sb.append(s)       누적방법
   BinaryOperator<A>     combiner();         // (sb1, sb2) -> sb1.append(sb2) 결합방법(병렬)
   Function<A, R>        finisher();         // sb -> sb.toString()           최종변환
   Set<Characteristics> characteristics(); // 컬렉터의 특성이 담긴 Set을 반환
    ...
}
```

▶ Collectors클래스는 다양한 기능의 컬렉터(Collector를 구현한 클래스)를 제공

- 변환 - mapping(), toList(), toSet(), toMap(), toCollection(), …
- 통계 - counting(), summingInt(), averagingInt(), maxBy(), minBy(), summarizingInt(), …
- 문자열 결합 - joining()
- 리듀싱 - reducing()
- 그룹화와 분할 - groupingBy(), partitioningBy(), collectingAndThen()

2.8 Collectors의 메서드(1/4)

▶ 스트림을 컬렉션으로 변환 – toList(), toSet(), toMap(), toCollection()

```
List<String> names = stuStream.map(Student::getName)  // Stream<Student>→Stream<String>
                              .collect(Collectors.toList()); // Stream<String>→List<String>
ArrayList<String> list = names.stream()
  .collect(Collectors.toCollection(ArrayList::new)); // Stream<String>→ArrayList<String>

Map<String,Person> map = personStream
  .collect(Collectors.toMap(p->p.getRegId(), p->p));// Stream<Person>→Map<String,Person>
```

▶ 스트림의 통계정보 제공 – counting(), summingInt(), maxBy(), minBy(), …

```
long count = stuStream.count();
long count = stuStream.collect(counting()); // Collectors.counting()
```

```
long totalScore = stuStream.mapToInt(Student::getTotalScore).sum();  // IntStream의 sum()
long totalScore = stuStream.collect(summingInt(Student::getTotalScore));
```

```
OptionalInt topScore = studentStream.mapToInt(Student::getTotalScore).max();
Optional<Student> topStudent = stuStream
                          .max(Comparator.comparingInt(Student::getTotalScore));
Optional<Student> topStudent = stuStream
             .collect(maxBy(Comparator.comparingInt(Student::getTotalScore)));
```

2.8 Collectors의 메서드(2/4)

▶ 스트림을 리듀싱 - reducing()

```
Collector reducing(BinaryOperator<T> op)
Collector reducing(T identity, BinaryOperator<T> op)
Collector reducing(U identity, Function<T,U> mapper, BinaryOperator<U> op) // map+reduce
```

```
IntStream intStream = new Random().ints(1,46).distinct().limit(6);

OptionalInt       max = intStream.reduce(Integer::max);
Optional<Integer> max = intStream.boxed().collect(reducing(Integer::max));
```

```
long sum = intStream.reduce(0, (a,b) -> a + b);
long sum = intStream.boxed().collect(reducing(0, (a,b)-> a + b));
```

```
int grandTotal = stuStream.map(Student::getTotalScore).reduce(0, Integer::sum);
int grandTotal = stuStream.collect(reducing(0, Student::getTotalScore, Integer::sum));
```

▶ 문자열 스트림의 요소를 모두 연결 - joining()

```
String studentNames = stuStream.map(Student::getName).collect(joining());
String studentNames = stuStream.map(Student::getName).collect(joining(",")); // 구분자
String studentNames = stuStream.map(Student::getName).collect(joining(",", "[", "]"));
String studentInfo  = stuStream.collect(joining(",")); // Student의 toString()으로 결합
```

2.8 Collectors의 메서드(3/4)

▶ 스트림의 요소를 2분할 - partitioningBy()

```
Collector partitioningBy(Predicate predicate)
Collector partitioningBy(Predicate predicate, Collector downstream)
```

```
Map<Boolean, List<Student>> stuBySex = stuStream
                    .collect(partitioningBy(Student::isMale)); // 학생들을 성별로 분할
List<Student> maleStudent   = stuBySex.get(true);  // Map에서 남학생 목록을 얻는다.
List<Student> femaleStudent = stuBySex.get(false); // Map에서 여학생 목록을 얻는다.
```

```
Map<Boolean, Long> stuNumBySex = stuStream
                    .collect(partitioningBy(Student::isMale, counting())); // 분할 + 통계
System.out.println("남학생 수 :"+ stuNumBySex.get(true));  // 남학생 수 :8
System.out.println("여학생 수 :"+ stuNumBySex.get(false)); // 여학생 수 :10
```

```
Map<Boolean, Optional<Student>> topScoreBySex = stuStream                    // 분할 + 통계
  .collect(partitioningBy(Student::isMale, maxBy(comparingInt(Student::getScore))));
System.out.println("남학생 1등 :"+ topScoreBySex.get(true)); // 남학생 1등 :Optional[[나자바,남, 1, 1,300]]
System.out.println("여학생 1등 :"+ topScoreBySex.get(false)); //여학생 1등 :Optional[[김지미,여, 1, 1,250]]
```

```
Map<Boolean, Map<Boolean, List<Student>>> failedStuBySex = stuStream       // 다중 분할
.collect(partitioningBy(Student::isMale,                  // 1. 성별로 분할(남/녀)
         partitioningBy(s -> s.getScore() < 150)));   // 2. 성적으로 분할(불합격/합격)
List<Student> failedMaleStu   = failedStuBySex.get(true).get(true);
List<Student> failedFemaleStu = failedStuBySex.get(false).get(true);
```

2.8 Collectors의 메서드(4/4)

▶ 스트림의 요소를 그룹화 - groupingBy()

```
Collector groupingBy(Function classifier)
Collector groupingBy(Function classifier, Collector downstream)
Collector groupingBy(Function classifier, Supplier mapFactory, Collector downstream)
```

```
Map<Integer, List<Student>> stuByBan = stuStream                    // 학생을 반별로 그룹화
              .collect(groupingBy(Student::getBan, toList()));  // toList() 생략가능
```

```
Map<Integer, Map<Integer, List<Student>>> stuByHakAndBan = stuStream  // 다중 그룹화
              .collect(groupingBy(Student::getHak,                    // 1. 학년별 그룹화
                       groupingBy(Student::getBan)                    // 2. 반별 그룹화
               ));
```

```
Map<Integer, Map<Integer, Set<Student.Level>>> stuByHakAndBan = stuStream
.collect(
     groupingBy(Student::getHak, groupingBy(Student::getBan,   // 다중 그룹화(학년별, 반별)
          mapping(s-> {   // 성적등급(Level)으로 변환.  List<Student> → Set<Student.Level>
               if     (s.getScore() >= 200) return Student.Level.HIGH;
               else if(s.getScore() >= 100) return Student.Level.MID;
               else                         return Student.Level.LOW;
          } , toSet())  // mapping()              // enum Level { HIGH, MID, LOW }
     ))  // groupingBy()
); // collect()
```

2.9 Collector 구현하기

▶ Collector인터페이스를 구현하는 클래스를 작성

```
public interface Collector<T, A, R> {  // T(요소)를 A에 누적한 다음, 결과를 R로 변환해서 반환
   Supplier<A>           supplier();         // 결과를 저장할 공간(A)을 제공
   BiConsumer<A, T>      accumulator();      // 스트림의 요소(T)를 수집(collect)할 방법을 제공
   BinaryOperator<A>     combiner();         // 두 저장공간(A)을 병합할 방법을 제공(병렬 스트림)
   Function<A, R>        finisher();         // 최종변환(A → R). 변환할 필요가 없는 경우, x->x
   Set<Characteristics> characteristics(); // 컬렉터의 특성이 담긴 Set을 반환
    ...
}
```

▶ 컬렉터가 수행할 작업의 속성 정보를 제공 – characteristics()

```
Characteristics.CONCURRENT          병렬로 처리할 수 있는 작업
Characteristics.UNORDERED           스트림의 요소의 순서가 유지될 필요가 없는 작업
Characteristics.IDENTITY_FINISH     finisher()가 항등 함수인 작업
```

```
public Set<Characteristics> characteristics() {
     return Collections.unmodifiableSet(EnumSet.of(
               Collector.Characteristics.CONCURRENT, Collector.Characteristics.UNORDERED
            ));
}
```

```
Set<Characteristics> characteristics() {
   return Collections.emptySet(); // 지정할 특성이 없으면 빈 Set을 반환
}
```

2.9 Collector 구현하기 – example

▶ 문자열 스트림의 모든 요소를 연결하는 컬렉터 – ConcatCollector

```
class ConcatCollector implements Collector<String, StringBuilder, String> {
    public Supplier<StringBuilder> supplier() {
        return () -> new StringBuilder();  // return StringBuilder::new;
    }

    public BiConsumer<StringBuilder, String> accumulator() {
        return (sb,s) -> sb.append(s);
    }

    public Function<StringBuilder, String> finisher() {
        return sb -> sb.toString();
    }

    public BinaryOperator<StringBuilder> combiner() {
        return (sb, sb2) -> sb.append(sb2);
    }

    public Set<Characteristics> characteristics() {
        return Collections.emptySet();
    }
}
```

```
String[] strArr = {"aaa","bbb","ccc" };
// supplier()
StringBuffer sb = new StringBuffer();

for(String tmp : strArr)
    sb.append(tmp); // accumulator()
// finisher()
String result = sb.toString();
```

```
public static void main(String[] args) {
    String[] strArr = { "aaa","bbb","ccc" };
    Stream<String> strStream = Stream.of(strArr);
    String result = strStream.collect(new ConcatCollector());
    System.out.println("result="+result); // result=aaabbbccc
}
```

2.10 스트림의 변환(1/2)

from	to	변환 메서드
1. 스트림 → 기본형 스트림		
Stream<T>	IntStream LongStream DoubleStream	mapToInt(ToIntFunction<T> mapper) mapToLong(ToLongFunction<T> mapper) mapToDouble(ToDoubleFunction<T> mapper)
2. 기본형 스트림 → 스트림		
IntStream LongStream DoubleStream	Stream<Integer> Stream<Long> Stream<Double>	boxed()
	Stream<U>	mapToObj(DoubleFunction<U> mapper)
3. 기본형 스트림 → 기본형 스트림		
IntStream LongStream DoubleStream	LongStream DoubleStream	asLongStream() asDoubleStream()
4. 스트림 → 부분 스트림		
Stream<T> IntStream	Stream<T> IntStream	skip(long n) limit(long maxSize)
5. 두 개의 스트림 → 스트림		
Stream<T>, Stream<T>	Stream<T>	concat(Stream<T> a, Stream<T> b)
IntStream, IntStream	IntStream	concat(IntStream a, IntStream b)
LongStream, LongStream	LongStream	concat(LongStream a, LongStream b)
DoubleStream, DoubleStream	DoubleStream	concat(DoubleStream a, DoubleStream b)
6. 스트림의 스트림 → 스트림		
Stream<Stream<T>>	Stream<T>	flatMap(Function mapper)
Stream<IntStream>	IntStream	flatMapToInt(Function mapper)
Stream<LongStream>	LongStream	flatMapToLong(Function mapper)
Stream<DoubleStream>	DoubleStream	flatMapToDouble(Function mapper)

2.10 스트림의 변환(2/2)

from	to	변환 메서드
7. 스트림 ↔ 병렬 스트림		

Stream<T>
IntStream
LongStream
DoubleStream
	Stream<T>
	IntStream
	LongStream
	DoubleStream
		parallel() // 스트림 → 병렬 스트림
		sequential() // 병렬 스트림 → 스트림

8. 스트림 → 컬렉션		
Stream<T> IntStream LongStream DoubleStream	Collection<T>	collect(Collectors.toCollection(Supplier factory))
	List<T>	collect(Collectors.toList())
	Set<T>	collect(Collectors.toSet())
9. 컬렉션 → 스트림		
Collection<T> List<T> Set<T>	Stream<T>	stream()
10. 스트림 → Map		
Stream<T> IntStream LongStream DoubleStream	Map<K,V>	collect(Collectors.toMap(Function key, Function value)) collect(Collectors.toMap(Function, Function, BinaryOperator)) collect(Collectors.toMap(Function, Function, BinaryOperator merge, Supplier mapSupplier))
11. 스트림 → 배열		
Stream<T>	Object[]	toArray()
	T[]	toArray(IntFunction<A[]> generator)
IntStream LongStream DoubleStream	int[] long[] double[]	toArray()

감사합니다.

Q & A

castello@naver.com

www.codechobo.com